

7N-61-TM
RNR-91-025

Automatic Translation of Fortran Programs to Multiprecision
David H. Bailey
RNR Technical Report RNR-91-025
October 25, 1991



National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California 94035

ARC 275 (Rev Feb 81)

Automatic Translation of Fortran Programs to Multiprecision
David H. Bailey
RNR Technical Report RNR-91-025
October 25, 1991

Abstract

Previously the author developed a package of Fortran subroutines to perform a variety of arithmetic operations and transcendental functions on floating point numbers of arbitrarily high precision. This package is in some cases over 200 times faster than that of certain other packages that have been developed for this purpose.

However, as with other such packages, manually converting a program to use the author's routines is a tedious and error-prone process. To facilitate such conversions, the author has developed a translator program. By means of source directives (special comments), the user declares the precision level and specifies which variables in each program unit are to be treated as multiprecision. The translator program reads this source program and outputs a program with the appropriate multiprecision subroutine calls.

This translator program supports multiprecision integer, real and complex data types. The required array space for multiprecision data types is automatically allocated. In the evaluation of computational expressions, all of the usual conventions for operator precedence and mixed mode operations are upheld. Furthermore, most of the Fortran-77 intrinsics, such as ABS, MOD, NINT, COS, EXP are supported and produce true multiprecision values.

The author is with the NAS Applied Research Branch, NASA Ames Research Center, Moffett Field, CA 94035. E-mail: dbailey@nas.nasa.gov.

0. Introduction

The author's MPFUN package is a suite of Fortran subroutines that perform arithmetic on floating point numbers of arbitrarily high precision. It is described in detail in [1].

MPFUN routines are available to perform the four basic arithmetic operations between MP numbers, to compare MP numbers, to produce the integer and fractional parts, to produce a random MP number and to perform binary to decimal and decimal to binary conversion. Some higher level routines sort MP numbers; perform complex arithmetic; compute square roots, cube roots, n -th powers, n -th roots, and π ; evaluate the functions exp, log, cos, sin, cosh, sinh, inverse cos and sin; find the real or complex roots of polynomials; and find integer relations in real vectors.

Computations on large integers can also be efficiently performed using this package by setting the working precision level two or three words higher than the largest integer that will be encountered (including products).

One key feature of the MPFUN package is that it was written with a vector supercomputer or RISC floating point computer in mind from the beginning. Virtually all inner loops are vectorizable and employ floating point operations, which have the highest performance on these systems. As a result, MPFUN exhibits excellent performance on such systems.

Another distinguishing feature of the MPFUN package is its usage of advanced algorithms. For many functions, both a "basic" and an "advanced" routine are provided. The advanced routines employ advanced algorithms and exhibit superior performance for extra-high precision (i.e. above about 1000 digit) calculations. For example, an advanced multiplication routine is available that employs a fast Fourier transform (FFT), and routines implementing the new Borwein quadratically convergent algorithms for exp and log are also provided.

1. An Automatic Multiprecision Translator

Conversion of a conventional scientific application program to use the MPFUN routines is generally straightforward, but it is often tedious and error prone. For example, if the slightest error is made in any of the arguments to the many subroutine calls, not only will the results be in error, but the program may abort with little information to guide the programmer. As a result of these difficulties, few serious scientific programs have yet been converted to use the MPFUN routines. Similar difficulties have plagued programmers who have attempted to use other multiprecision systems, such as Brent's package [4].

To facilitate such conversions, the author has developed a translator program that accepts as input a conventional Fortran-77 program to which has been added certain special comments that declare the desired level of precision and declare certain variables in each program unit to be treated as multiprecision. This translator then parses the input code and generates an output program that has all of the calls to the appropriate MPFUN routines. This output program may then be compiled and linked with the MPFUN package for execution.

This translation program allows one to extend the Fortran-77 language with the data

types `MULTIP INTEGER`, `MULTIP REAL` and `MULTIP COMPLEX`. These data types can be used for integer, floating point or complex numbers of an arbitrarily high but pre-specified level of precision. Variables in the input program may be declared to have one of these multiprecision types in the output program by placing directives (special comments) in the input file. In this way, the input file remains an ANSI Fortran-77 compatible program and can be run at any time using ordinary arithmetic on any Fortran system for comparison with the multiprecision equivalent.

This translator supports a large number of Fortran-77 constructs involving multiprecision variables, including all the standard arithmetic operators, mixed mode expressions, automatic type conversions, comparisons, logical `IF` statements (including `IF-THEN-ELSE` constructs), function calls, `READ` and `WRITE` statements and most of the Fortran intrinsics (i.e. `ABS`, `MOD`, `COS`, `EXP`, etc.). Storage is automatically allocated for multiprecision variables, including temporaries, and the required initialization for the `MPFUN` package is automatically performed.

This processor translates programs to use the standard `MP` routines from the author's `MPFUN` package. If one wishes to utilize this translator in connection with the extra-high precision routines of this package, which are designed for precision levels greater than about 1,000 digits, contact the author for instructions.

2. Operation of the Translator Program

This translator program should run on any Fortran-77 system that supports recursive subroutine references. On some systems, including Sun and IBM workstations, a minor source modification and/or a special compiler option must be enabled to permit this recursion. A test program is available to determine whether the translator is working correctly on a given system.

The translator is in effect a compiler in the sense that it identifies and analyzes every input statement. It develops a symbol table that contains type and dimension information for all variables used in a program unit. A number of Fortran statements, such as `DO`, `CONTINUE` and `OPEN` statements, are not modified by the translator. Most other types of statements are analyzed in detail, including type declarations, `IMPLICIT`, `COMMON`, `DIMENSION`, `PARAMETER`, `READ`, `WRITE` and `CALL` statements, as well as all assignment statements.

If any input statement is modified or translated, the original statement is included in the output file as a comment, starting with the string `CMP>`. The comment `CMP<` is placed after the translated lines. Warnings and error messages are also written in the output file. Warnings are issued as comments starting with `CMP*`. Fatal error messages start with `***`. When a fatal error is detected, the message is output on the output file, and processing is terminated. Thus to make sure that the translation of an input program was successful, check the end of the output file to make sure there is no fatal error message. It is also strongly recommended that the output program be scanned for `CMP*` warning messages before it is compiled and executed.

3. Basic Instructions for Usage

These data type abbreviations will be used hereafter in this paper:

IN	Integer
SP	Single precision real
DP	Double precision real
CO	Single precision complex
DC	Double precision complex (non-ANSI extension of Fortran-77)
MPI	Multiprecision integer
MPR	Multiprecision real
MPC	Multiprecision complex
MP	Denotes the three multiprecision types collectively

A MP statement will be defined as a statement that has at least one MP variable. A MP program unit will be defined as a program unit with at least one MP variable.

At the beginning of a file containing a conventional Fortran-77 code to be translated, before any program or subroutine statement, a directive (i.e. special comment) of the following form must be inserted:

```
CMP+ PRECISION LEVEL 120
```

This denotes that the maximum precision level to be employed in this program is 120 digits. Only one such declaration is allowed in a single file, and Fortran-77 files whose translated routines later will be linked together must have equivalent precision level declarations. This directive must precede any of the other **CMP+** directives to be described below.

Variables in a subprogram of the input Fortran-77 program file that are to be treated as MP by the translator program are declared by one or more MP type directives such as the following:

```
CMP+ MULTIP INTEGER IA, IPR, MODUL
```

```
CMP+ MULTIP REAL SUM, TL, X, Y
```

```
CMP+ MULTIP COMPLEX W, ZAB
```

A MP variable must be declared in a MP type directive prior to any appearance of that variable in the subprogram, including any appearance in a type declaration, **DIMENSION** or **COMMON** statement. An exception to this rule is that MP variable names appearing in the argument list of a **FUNCTION** or **SUBROUTINE** statement may be afterwards declared. However, if the function name of a function subprogram is to have a MP type, this name must be declared with a MP type directive immediately preceding the **FUNCTION** statement. The dimensions for a MP variable are not included in the MP type directive — they will be taken from the standard type declaration, **DIMENSION** or **COMMON** statement where these dimensions are defined in the original program.

4. The SAFE and FAST Options

Expressions involving MP variables and constants are evaluated using the operator precedence conventions of Fortran-77, and using predictable extensions of the Fortran-77 mixed mode conventions. There are two options for the evaluation of mixed mode operations: FAST and SAFE. The difference between these conventions may be seen with the following example, where A and B are MPR and N is an ordinary integer variable:

$$B = A + 1.DO / N$$

With the FAST option, the subexpression $1.DO / N$ is evaluated using DP arithmetic, and the result temporary has type DP. With the SAFE option, which is the default, $1.DO / N$ is performed using MP arithmetic, and the result temporary has type MPR. As the name signifies, the FAST option produces somewhat more efficient translated code, but it may also give unexpectedly inaccurate answers, for instance if N in the above example has the value 7.

An exception to the SAFE option is in the argument lists of subroutine calls or non-intrinsic function references. Expressions appearing in these lists are always evaluated using the FAST option, since this corresponds more closely to the Fortran convention that most users expect. Thus in the statement

$$B = 3 * FUN (N - 1, A)$$

the subexpression $N - 1$ is always evaluated using ordinary integer arithmetic, and the result temporary has type IN.

The user may switch between these options by inserting one of the following directives in the declaration section of any program unit.

```
CMP+ MIXED MODE FAST
CMP+ MIXED MODE SAFE
```

For the operators $+$ $-$ $*$ $/$, Tables 1 and 2 give the types of results with these two options. Table 3 lists the argument types and results defined for the $**$ operator. In Table 3, if a particular combination is not listed, or if its position in the table is blank, then it is not defined. Comparison operations (i.e. $.EQ.$, $.LT.$, etc.), where one or both of the operands are MP, are permitted both in logical IF statements and in logical assignment statements. If one of the operands has type CO, DC or MPC, only $.EQ.$ and $.NE.$ comparisons are permitted.

5. Multiprecision Constants

With the SAFE option, all IN constants appearing in MP statements are considered MPI constants and are converted to full precision, and all SP or DP constants in MP statements are considered MPR constants and are converted to full precision.

With the FAST option, IN, SP and DP constants are recognized and treated as such by the translator — they are merely passed unchanged to the output program and are

Arg. 1 / Arg. 2	IN	SP	DP	CO	DC	MPI	MPR	MPC
IN	IN	SP	DP	CO	DC	MPI	MPR	MPC
SP	SP	SP	DP	CO	DC	MPR	MPR	MPC
DP	DP	DP	DP	DC	DC	MPR	MPR	MPC
CO	CO	CO	CO	CO	DC	MPC	MPC	MPC
DC	DC	DC	DC	DC	DC	MPC	MPC	MPC
MPI	MPI	MPR	MPR	MPC	MPC	MPI	MPR	MPC
MPR	MPR	MPR	MPR	MPC	MPC	MPR	MPR	MPC
MPC	MPC	MPC	MPC	MPC	MPC	MPC	MPC	MPC

Table 1: Results of Mixed Mode Arithmetic Operations with the FAST option

Arg. 1 / Arg. 2	IN	SP	DP	CO	DC	MPI	MPR	MPC
IN	MPI	MPR	MPR	MPC	MPC	MPI	MPR	MPC
SP	MPR	MPR	MPR	MPC	MPC	MPR	MPR	MPC
DP	MPR	MPR	MPR	MPC	MPC	MPR	MPR	MPC
CO	MPC	MPC	MPC	MPC	MPC	MPC	MPC	MPC
DC	MPC	MPC	MPC	MPC	MPC	MPC	MPC	MPC
MPI	MPI	MPR	MPR	MPC	MPC	MPI	MPR	MPC
MPR	MPR	MPR	MPR	MPC	MPC	MPR	MPR	MPC
MPC	MPC	MPC	MPC	MPC	MPC	MPC	MPC	MPC

Table 2: Results of Mixed Mode Arithmetic Operations with the SAFE option (default)

Arg. 1	Arg. 2	Result	
		FAST	SAFE
IN	IN	IN	MPI
IN or SP	SP	SP	MPR
IN, SP or DP	DP	DP	MPR
IN, SP or CO	CO	CO	
IN, SP, DP, CO or DC	DC	DC	
IN	MPI	MPI	MPI
IN, SP or DP	MPR	MPR	MPR
CO	IN	CO	MPC
CO	SP	CO	
CO	DP or DC	DC	
DC	IN	DC	MPC
DC	SP, DP, CO or DP	DC	
MPI	IN or MPI	MPI	MPI
MPI	SP, DP or MPR	MPR	MPR
MPR	IN, SP, DP, MPI or MPR	MPR	MPR
MPC	IN	MPC	MPC

Table 3: Defined Combinations for the ** Operator

converted to binary by the underlying Fortran system. For modest sized whole numbers and exact binary fractions, these constants are converted exactly and produce accurate results when they appear in expressions with MP variables. However, SP or DP constants that cannot be precisely converted (i.e. 1.01D0), or IN, SP or DP constants that have more significant digits than can be exactly accommodated in these datatypes, may result in inaccurate MP calculations.

To avoid such difficulties with the FAST option, the user may explicitly specify that a constant in the input program will be treated as a MP constant for the output program. This is done by appending +0 to the constant, as in the following examples:

```
3+0
-12345678901234567890+0
3.141592653589793+0
1.2345678901234567890D-13+0
```

The first two constants have type MPI, and the last two have type MPR. Embedded blanks are allowed anywhere in these constants, including before and after the plus sign. MP constants must appear in a context where the plus operation would actually be performed between the two components of the MP constant if interpreted according to the standard Fortran rules for evaluating expressions. For example, the expression N*12345+0 is not treated as containing a MP constant. Write this as N*(12345+0) instead. MP constants are recognized as such only in MP statements.

There is no definition of this sort for MPC constants, but MPC constants may be defined by using the special conversion function DPCMPL (see section 6), where the two arguments are MPR constants.

MP constants may be defined symbolically using PARAMETER statements. The parameter assignment expression for a MP variable may reference previously defined MP and non-MP parameters, and it may also include intrinsic function references. All such assignments are performed upon entry to the program unit the first time it is called.

6. Intrinsic Functions

Table 4 lists Fortran intrinsic functions that are supported by this translator with MP arguments. References to these functions will be automatically translated to call the appropriate routine(s) from the MPFUN package, provided the argument(s) is(are) of the appropriate MP type. If the SAFE option is in effect, non-MP arguments are first converted to MP, so that MP results are always returned. If the user requires either a function not listed here or a function with an argument type not listed here, contact the author.

Note in Table 4 that the "conversion" intrinsics of Fortran-77, namely INT, CMPLX, DBLE, DCMPLX and REAL, return types IN, CO, DP, DC and SP, respectively, even though the arguments have MP types. This is in keeping with the conventions of Fortran-77. If one wishes to truncate a MPR number to MPI, form a MPC number from two MPR numbers, or extract the MPR real and imaginary components of a MPC number, one of the special functions in Table 5 should be used instead. These functions are not defined for ordinary

Function	Arg. 1	Arg. 2	Result
ABS	MPI	MPR	MPI
	MPR		MPR
	MPC		MPR
ACOS	MPR		MPR
AINT	MPR		MPR
ANINT	MPR		MPR
ASIN	MPR		MPR
ATAN	MPR		MPR
ATAN2	MPR		MPR
CMPLX	MPC		CO
CONJG	MPC		MPC
COS	MPR		MPR
COSH	MPR		MPR
DBLE	MPI		DP
	MPR		DP
	MPC		DP
DCMPLX	MPC		DC
EXP	MPR		MPR
INT	MPI	MPI	IN
	MPR		IN
	MPC		IN
LOG	MPR		MPR
LOG10	MPR		MPR
MAX	MPI		MPI
	MPR		MPR
MIN	MPI		MPI
	MPR		MPR
MOD	MPI		MPI
	MPR		MPR
NINT	MPI		MPI
	MPR		MPR
REAL	MPI		SP
	MPR		SP
	MPC		SP
SIN	MPR	MPR	MPR
SINH	MPR		MPR
SQRT	MPR		MPR
	MPC		MPC
TAN	MPR		MPR
TANH	MPR		MPR

Table 4: Fortran Intrinsics Supported with MP Arguments

SP, DP, CO or DC arguments in the translated program (although they may be in the input program). Thus, for example, DPREAL cannot be used to convert a DP number to MPR. Type conversions such as this can be performed either by simple assignment statements, or by defining an external MP function.

To preserve comparable functionality between an input Fortran-77 program that uses one of these four special conversion functions and the output MP program, equivalent SP or DP functions should be included in the input program as user-defined subprograms (not as statement functions). Table 6 has some examples of equivalent definitions for these functions that use DP and DC data types. If your program uses ordinary SP and CO data types instead, these definitions need to be changed accordingly.

Do not place any MP directives in these function subprograms. If another subprogram references one of these functions, it should declare the argument and function names to be of the appropriate types, corresponding to the function definitions. However, the names MPINT, DPCMPL, DPREAL and DPIMAG do not need to be declared with MP type directives in the program units where they are referenced. In the output program, MP results will be automatically be returned with types according to Table 5.

With the FAST option, non-MP arguments to intrinsic functions appearing in MP statements are passed without change to the non-MP intrinsic functions. For non-MP arguments the translator recognizes the following "generic" intrinsic function names and assigns result types according to argument types, in accordance with the standard Fortran conventions:

ABS, ACOS, AINT, AIMAG, ANINT, ASIN, ATAN, ATAN2, CHAR, CMPLX, CONJG,
COS, COSH, DBLE, DCMPLX, DIM, DIMAG, DREAL, EXP, ICHAR, INDEX, INT, LEN,
LOG, LOG10, MAX, MIN, MOD, NINT, REAL, SIGN, SIN, SINH, SQRT, TAN, TANH.

Note that this list does not include type-specific variants such as IABS and AMOD. Convert specific names such as these to the generic name in above list, or else include the specific function name in a conventional type statement at the beginning of the subprogram.

7. Other Special Functions and Constants

In many instances where COS or SIN is referenced, both are actually required for the same argument. Since MPCSSN, the MPFUN routine that evaluates COS or SIN for a MPR argument, actually returns both in a single call, it is preferable for the input DP code to have an analogous design. Thus it is recommended that input code which references COS and SIN be changed to reference a subroutine DPCSSN, which may be written as follows for DP data. If your code uses the SP data type, this definition should be changed accordingly.

Function	Arg. 1	Arg. 2	Result
MPINT	MPR		MPI
DPCMPL	MPR	MPR	MPC
DPREAL	MPC		MPR
DPIMAG	MPC		MPR

Table 5: Special MP Conversion Functions

```

FUNCTION MPINT (X)
DOUBLE PRECISION X
MPINT = INT (X)
RETURN
END

```

```

FUNCTION DPCMPL (A, B)
DOUBLE COMPLEX DPCMPL
DOUBLE PRECISION A, B
DPCMPL = DCMPLX (A, B)
RETURN
END

```

```

FUNCTION DPREAL (C)
DOUBLE PRECISION DPREAL
DOUBLE COMPLEX C
DPREAL = DBLE (C)
RETURN
END

```

```

FUNCTION DPIMAG (C)
DOUBLE PRECISION DPIMAG
DOUBLE COMPLEX C
DPIMAG = DIMAG (C)
RETURN
END

```

Table 6: DP Equivalents of the Special Conversion Functions

```

SUBROUTINE DPCSSN (A, X, Y)
DOUBLE PRECISION A, X, Y
X = COS (A)
Y = SIN (A)
RETURN
END

```

The analogous DP subroutine name recognized for the hyperbolic functions COSH and SINH is DPCSSH.

Another operation of this nature is root extraction, i.e. $B = A ** (1.DO / N)$, for which the efficient routine MPNRT exists in the MPFUN package. Thus it is recommended that any code in the input program that performs root extraction using the ****** operator be changed to reference the function DPNRT instead, i.e. $B = DPNRT (A, N)$, where DPNRT may be written as follows for DP data. If the SP data type is used instead, this definition should be changed accordingly.

```

FUNCTION DPNRT (A, N)
DOUBLE PRECISION A, DPNRT
DPNRT = A ** (1.DO / N)
RETURN
END

```

One additional special function that many users may find useful produces pseudorandom MPR numbers. The routine MPRAND in the MPFUN package generates pseudorandom numbers uniformly in the range (0,1). To access this routine by means of the translator, one references the special function DPRAND. This function has no arguments — one references it by means of statements such as $A = 3 * DPRAND ()$. It is not possible to write a completely equivalent DP version of this routine. However, the basic pseudorandom number functionality can be reproduced by means of a simple routine such as the one shown in Table 7.

The sample program definitions listed above for DPCSSN, DPCSSH, DPNRT and DPRAND, like those for the special conversion intrinsics mentioned in section 6, are only for the purpose of providing comparable functionality when the input program is run with ordinary arithmetic. Do not place any MP directives in any of these subprograms. If another subprogram references either DPNRT or DPRAND, it should declare the function name to be of the appropriate type (DP in the examples above). However, the names DPNRT and DPRAND do not need to be declared with a MP type directive in program units that reference them.

The constants $\log 2 = 0.69314\dots$, $\log 10 = 2.30258\dots$ and $\pi = 3.14159\dots$ are computed in the program initialization and are available in any subprogram that contains MP variables. These values may be referenced by the user by means of the special variable names DPL02, DPL10 and DPPIC. Whenever any of these names appears in a statement, this translator substitutes the MP value. Any subprogram that references any of these

```

      FUNCTION DPRAND ()
C
C   This routine returns a pseudorandom DP floating number nearly uniformly
C   distributed between 0 and 1 by means of a linear congruential scheme.
C   228 pseudorandom numbers with 30 bits each are returned before repeating.
C
      IMPLICIT DOUBLE PRECISION (A-H, O-Z)
      PARAMETER (F7 = 78125.D0, R30 = 0.5D0 ** 30, T30 = 2.D0 ** 30)
      SAVE SD
      DATA SD/314159265.D0/
C
      T1 = F7 * SD
      T2 = AINT (R30 * T1)
      SD = T1 - T30 * T2
      DPRAND = R30 * SD
C
      RETURN
      END

```

Table 7: Suggested DP Equivalent of the Special Function DPRAND

parameters must declare it to be DP and must set its DP value in a parameter statement.
Example:

```

      DOUBLE PRECISION DPPIC
      PARAMETER (DPPIC = 3.141592653589793D0)

```

This parameter statement will be ignored in the output program, and the MP value will be used instead. The names DPL02, DPL10 and DPPIC do not need to be declared with a MP type directive. Do not attempt to define any of these values by means of assignments or function calls.

8. Input and Output of MP Numbers

MP variables may appear in READ or WRITE statements with the following two special forms:

```

      WRITE (6, *) VAR1, VAR2(I), VAR3(I,J)
      READ (11) VAR1, VAR2, VAR3

```

Either form may be a READ or WRITE, but neither may employ implied DO loops — convert these to explicit DO loops instead. The unit numbers may be integer variables instead of integer constants. Non-MP variables and constants may be included in the list, in which case they are handled using ordinary Fortran I/O.

The first form is used for input and output of individual MP numbers (not entire unsubscripted arrays) in ordinary decimal form. The digits of the number may span more than one line. A comma at the end of the last line denotes the end of a MP number. Input lines may not exceed 120 characters in length, but embedded blanks are allowed anywhere. The exponent is optional in an input number, but if present it must appear first, as in the following example:

```
10 ^ -4 x 3.14159 26535 89793 23846 26433 83279
50288 41971 69399 37510,
```

MPC numbers are input or output as two consecutive MPR numbers. The output of a MP write operation is in the correct form for a subsequent MP read operation. By default, all digits of a MP number are output. The user can control the number of digits output by including a directive such as

```
CMP+ OUTPUT PRECISION 200
```

in the declaration section of any subprogram. It remains in effect until the end of file or until another such directive is encountered.

The second form of READ/WRITE statement above is used to perform binary I/O of entire MP arrays. Subscripted variables are not allowed here.

9. Accessing the "Machine Epsilon"

Many MP programs need to control the "machine epsilon" for performing comparisons. Further, compatibility needs to be maintained with the original program. To this end, the user can employ a special constant with name DPEPS, which must be set to a DP value with a parameter statement in the subprogram that references it. Example:

```
DOUBLE PRECISION DPEPS
PARAMETER (DPEPS = 1D-16)
```

Whenever this name appears in a subprogram that contains MP variables, the translator substitutes the MP "epsilon" value, which by default is 10^{7-D} , where D is the number of digits of precision specified with the precision level directive mentioned above. DPEPS does not need to be declared with a MP type directive. The MP epsilon value may be modified (independent of the precision level directive) by inserting a directive such as

```
CMP+ EPSILON 1E-200
```

in the declaration section of any subprogram (for instance, immediately following the parameter statement in which DPEPS is defined). It remains in effect until the end of file or until another such directive is encountered.

10. Accessing the Precision Level

Some MP programs may need to access the precision level — for example so that the first dimension of MP arrays (which is the number of mantissa words plus 4) can be passed to a system routine. The number of mantissa words used for MP numbers (which is approximately one seventh the number of decimal digits) may be accessed in the special integer constant `MPNWP`. Whenever the name `MPNWP` occurs in a subprogram that contains MP variables, the translator automatically sets this parameter to the value of the `MPCOM1` parameter `NW`.

11. Single Precision Scratch Space for the MPFUN Package

The maximum amount of SP scratch space in common block `MPCOM3` (see the documentation for the MPFUN package [1]), cannot be determined in advance by the translator program. The MPFUN package allocates 1024 SP cells in this block, which for most programs is sufficient. If the “insufficient single precision scratch space” error is encountered during execution of the resulting MP program, place a directive of the form

```
CMP+ SCRATCH SPACE 2000
```

at the beginning of the input file, before any program or subroutine statement, after the precision level declaration. The number placed on this line should be at least the size mentioned in the error message.

12. Other Restrictions and Limitations

A number of other restrictions and limitations apply to programs processed by the translator. These restrictions are listed below. However, note that in almost every case there is a simple change that can be made to the input program to make it acceptable to this translation program, while retaining both its functionality and Fortran-77 compliance. Many of these restrictions are merely good programming practice.

Non-MP statements, i.e. statements that do not contain any MP variables, are not modified by this processor and are not subject to restrictions, except as noted in items 1 through 4.

1. Variable, common block and subroutine names beginning with `DP` and `MP` are reserved. Except as described in this paper, names with these prefixes should not be used in the input program.
2. A single `IMPLICIT` statement may be used to declare the initial letter(s) for only one type. A single `COMMON` statement may be used to declare only one common block, although a common block may contain any number of both MP and non-MP variables and arrays.
3. `DATA` statements and `FORMAT` statements may appear only after the end of the specification section of the program, i.e. only after type declaration, `DIMENSION` statements, `COMMON` statements, etc.

4. ENTRY, typed FUNCTION (i.e. INTEGER FUNCTION), assigned GOTO, arithmetic IF, READ or WRITE without parentheses, and PRINT statements are not allowed. Please replace these constructs, which in most cases are obsolescent, with more conventional alternatives: FUNCTION statements followed by type statements, normal subroutine calls, computed or ordinary GOTO statements, logical IF statements and normal READ or WRITE statements, respectively.
5. Statement functions may not be used to define MP functions. Convert these into MP function subprograms or subroutines.
6. MP variables may not appear in DATA statements. Convert these into parameter or assignment statements. For example, a DATA statement can be used to initialize a DP array, and then these DP values can be converted to MP and stored in a MP array by means of a loop containing an assignment statement.
7. The logical operators .NOT., .EQV. and .NEQV. may not appear in a MP statement. Rewrite such statements using .AND. and .OR. operators, or move such subexpressions to a separate statement.
8. Complex constants [i.e. (3., 2.)] may not appear in MP statements. Either use the intrinsic functions CMPLX or DCMLPX, or else assign such constants to CO or DC variables in separate statements.
9. A MP statement may not be the terminal line of a DO loop. Place the line number on a CONTINUE line immediately following the statement. If the line number is also the target of a GOTO, the DO loop must be changed to use a separate terminal line number.
10. Embedded blanks may not appear in Fortran keywords, line numbers, variable names, comparison operators and logical operators. Exceptions: DOUBLE PRECISION, DOUBLE COMPLEX, ELSE IF, END DO, END IF, GO TO are permitted.
11. Fortran keywords may not have mixed upper and lower case letters (i.e. FORmat).
12. Input code must be in the standard 72 column format. Comments up to 80 characters long are correctly copied to the output file.
13. REAL, DATA, DO or CALL must be followed by a blank (or an asterisk, as in REAL*8). Also, a blank must follow the line number in a DO statement.
14. If an integer constant is followed by a comparison or logical operator (i.e. .LE. or .AND.), they must be separated by a blank.
15. Tab characters are not allowed — convert these to blanks with a text editor.

On the other hand, this program will correctly process code with the following features, which do not comply with the Fortran-77 standard, provided the Fortran compiler being used also supports such constructs:

1. Lower case Fortran keywords, variable names, common block names and subroutine names are allowed.
2. Long variable names (up to 16 characters long) are permitted.
3. Character strings may be delimited with pairs of quotation marks ["] instead of apostrophes ['].
4. The double complex (DC) data type is supported, including DC intrinsics.
5. The data types INTEGER*4, REAL*8, etc. are supported. REAL*8 is interpreted as DP; COMPLEX*16 is interpreted as DC.
6. DO-ENDDO constructs are permitted.
7. Recursive subroutine calls are permitted.

13. Error Checking

Approximately 100 error conditions are checked by the translator program, and if any of these is encountered, an error message is output, together with the line number of the statement in the input file where the error was detected. An attempt has been made to cover the prohibited situations mentioned in this paper, as well as many violations of the standard rules of Fortran. In some cases, certain possible Fortran errors are not checked by the translator, because if they do occur, they will certainly be trapped when an attempt is made to compile the output program.

One example of an error condition that is checked by the translator is any type mismatch between the argument list of a reference to a subroutine or function and its definition (provided both are in the same file). Such errors can easily occur when, for example, a double precision constant is used as an argument, but the defining subprogram expects a multiprecision value. These errors can also occur if the function name of a MP function is not declared to be of type MP in the program unit where it is referenced.

Although this is certainly not a recommended programming practice, type mismatches between argument lists do exist in some working Fortran programs. For example, some codes pass a scratch array of type real to a subroutine when a complex scratch array is expected. Because in some cases it may be difficult to remove type mismatches from an existing code, and since the resulting code may work correctly anyway, a provision has been made for the translator to toggle type error trapping on and off. This is done by inserting one of the following directives in the declaration section of any program unit:

```
CMP+ TYPE ERROR ON
CMP+ TYPE ERROR OFF
```

It remains in effect until the end of file or until another such directive is encountered. When type error trapping is disabled with the OFF option, a non-fatal warning message is included in the output file for the programmer's information.

14. Examples of Translations

A number of fairly large programs have been successfully translated with this program. These include the Linpack benchmark [6], both a real and a complex FFT benchmark [2], a vortex analysis code [7], a Feigenbaum number calculation [5], an implementation of Ferguson's PSOS integer relation algorithm [3], and an implementation of the RSA public-key cryptosystem [8]. All appear to work correctly.

In most cases where the author had previously coded the application by hand using the MPFUN routines, the performance of the translated code (using the FAST option) is not significantly different. Thus it appears that in most cases there will not be a performance penalty for using the translator. Partly this is due to the fact that in translating arithmetic expressions, the translator program separately handles each of the many mixed mode cases, as opposed to merely handling all cases in a stock fashion.

Here are a few examples of how the translator handles various Fortran constructs. In the following, the FAST mixed mode option is in effect. IA and IB have type MPI, A, B, C and D have type MPR, and X and Y have type MPC. A number of temporary variables appear in the translated code. Those starting with MPD have type DP, those starting with MPJ have type MPI, those starting with MPL have type logical, those starting with MPM have type MPR, and those starting with MPZ have type MPC. The variable MPL02 is the MP translation of DPL02 (i.e. $\log 2$), MPEPS is the translation of DPEPS (i.e. the machine epsilon), and MPNW4 has the value MPNWP + 4 (i.e. the precision level plus 4).

An expression with subscripted MP variables:

```
C = C + 2.D0 * A(I) * (B(I) + 1.D0) ** 2
```

Translation:

```
CALL MPMULD (A(1,I), 2.D0, 0, MPM1)
CALL MPDMC (1.D0, 0, MPM3)
CALL MPADD (B(1,I), MPM3, MPM2)
CALL MPNPWR (MPM2, 2, MPM3)
CALL MPMUL (MPM1, MPM3, MPM2)
CALL MPADD (C, MPM2, MPM1)
CALL MPEQ (MPM1, C)
```

A statement with mixed DP, IN, MPI and MPR entities, and a LOG function reference:

```
D = IA + 3.D0 * LOG (C + 1)
```

Translation:

```

MPD1 = 1
CALL MPDMC (MPD1, 0, MPM2)
CALL MPADD (C, MPM2, MPM1)
CALL MPLOG (MPM1, MPL02, MPM2)
CALL MPMULD (MPM2, 3.DO, 0, MPM1)
CALL MPADD (IA, MPM1, MPM2)
CALL MPEQ (MPM2, D)

```

A reference to the user-defined external MP function DOT:

```

C = DOT (A, 25)

```

Translation:

```

CALL DOT (A, 25, MPM1)
CALL MPEQ (MPM1, C)

```

A complex square root reference, with the special function DPCMPL.

```

E = SQRT (DPCMPL (2.DO * S, S))

```

Translation:

```

CALL MPMULD (S, 2.DO, 0, MPM1)
CALL MPMMPC (MPM1, S, MPNW4, MPZ1)
CALL MPCSQR (MPNW4, MPZ1, MPZ2)
CALL MPCEQ (MPNW4, MPZ2, E)

```

An IF statement involving MP variables (including the special variable DPEPS), along with a WRITE statement:

```

IF (C .GT. 0.DO .AND. D .LT. DPEPS) WRITE (6, *) C

```

Translation:

```

CALL MPDMC (0.DO, 0, MPM1)
CALL MPCPR (C, MPM1, MPI1)
MPL1 = MPI1 .GT. 0
CALL MPCPR (D, MPEPS, MPI1)
MPL2 = MPI1 .LT. 0
MPL3 = MPL1 .AND. MPL2
IF (MPL3) THEN
CALL MPOUT (6, C, 120, MPA1)
ENDIF

```

Acknowledgments

The author wishes to acknowledge helpful comments and suggestions by W. Kahan of the University of California, Berkeley and by K. Briggs of the University of Melbourne, Australia.

References

- [1] Bailey, D. H., "MPFUN: A Portable High Performance Multiprecision Package", Technical Report RNR-90-022, NASA Ames Research Center, 1990. Submitted for publication.
- [2] Bailey, D. H., "A High Performance FFT Algorithm for Vector Supercomputers", *International Journal of Supercomputer Applications*, vol. 2 (Spring 1988), p. 82 - 87.
- [3] Bailey, D. H., and Ferguson, H. R. P., "Numerical Results on Relations Between Numerical Constants Using a New Algorithm", *Mathematics of Computation*, vol. 53 (October 1989), p. 649 - 656.
- [4] Brent, R. P., "A Fortran Multiple Precision Arithmetic Package", *ACM Transactions on Mathematical Software*, vol. 4 (1978), p. 57 - 70.
- [5] Briggs, Keith, "A Precise Calculation of the Feigenbaum Constants", *Mathematics of Computation*, vol. 57 (1991), p. 435 - 439.
- [6] Dongarra, J. J., "The Linpack Benchmark: An Explanation", *SuperComputing* (Spring 1988), p. 10 - 14.
- [7] Krasny, Robert, "Desingularization of Periodic Vortex Sheet Roll-Up", *Journal of Computational Physics*, vol. 65, no. 2 (August 1986), p. 292 - 313.
- [8] Rivest, R. L., Shamir, A., and Adleman, L., "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", *Communications of the ACM*, vol. 21 (1978), p. 120 - 126.